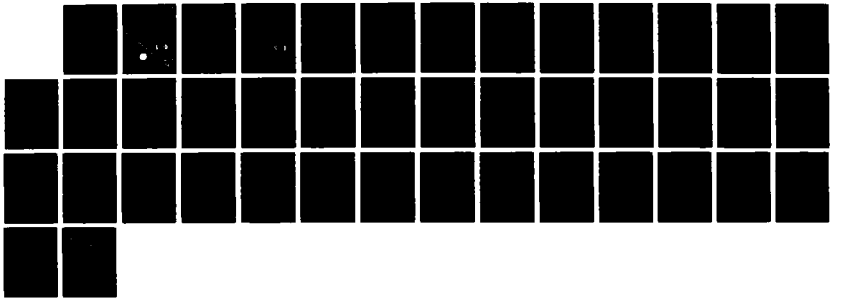
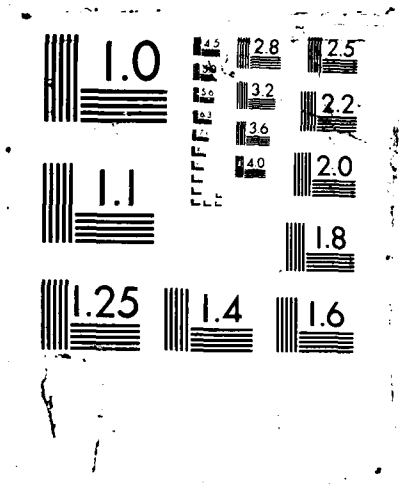
AD-A193 472 MPROLOG (MULTI VALUE PROLOG) MANUAL AND SOURCE CODE(U) 1/1
NEVADA UNIV LAS VEGAS DEPT OF COMPUTER SCIENCE AND
ELECTRICAL.. M FLATEBO FEB 88 CSR-88-003
UNCLASSIFIED ARO-24960.6-MA-REP DAAL03-87-G-0004 F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

ARO 24960.6-MA-REP

(2)

MPROLOG

Manual & Source Code*

by
Martin Flatebo

CSR-88-003

Department of Computer Science and Electrical Engineering

University of Nevada, Las Vegas

Las Vegas, Nevada 89154

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AR024960.6-MA-REP
6a. NAME OF PERFORMING ORGANIZATION University of Nevada 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION U. S. Army Research Office
6c. ADDRESS (City, State, and ZIP Code) Las Vegas, Nevada 89154	7b. ADDRESS (City, State, and ZIP Code) P. O. Box 12211 Research Triangle Park, NC 27709-2211
8a. NAME OF FUNDING / SPONSORING ORGANIZATION U. S. Army Research Office 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAL03-87-G-0004
8c. ADDRESS (City, State, and ZIP Code) P. O. Box 12211 Research Triangle Park, NC 27709-2211	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) MPROLOG: Manual & Source Code	
12. PERSONAL AUTHOR(S) Martin Flatebo	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) February 1988 15. PAGE COUNT 58
16. SUPPLEMENTARY NOTATION The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.	

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

Databases, Mprolog, Prolog, Logic Programming Language

19. ABSTRACT

This manual describes the use and implementation of a multi-valued Prolog called Mprolog. It is modeled after Prolog which uses horn clauses to store its collection of knowledge. Prolog is restricted by the fact that it is a two-valued logic and therefore can only examine information which is completely true or false. This is reasonable as long as the knowledge you want to represent does not contain uncertainties. If it does then you are not able to use the powerful programming language Prolog. What is needed is a multi-valued logic which allows information to be stored with uncertainties, and that is what Mprolog is all about. Mprolog is used to represent a new type of multi-value logic called minimal bounded fuzzy logic.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

(2)

MPROLOG

Manual & Source Code*

by
Martin Flatebo

CSR-88-003

*Funded by the Army Research Office under contract DAAL03 87-C 0004

Contents

Chapter 1

Mprolog

1.1 Introduction

This manual describes the use and implementation of a multi-valued Prolog called Mprolog. It is modeled after Prolog which uses horn clauses to store its collection of knowledge. Prolog is restricted by the fact that it is a two-valued logic and therefore can only examine information which is completely true or false. This is reasonable as long as the knowledge you want to represent does not contain uncertainties. If it does then you are not able to use the powerful programming language Prolog. What is needed is a multi-valued logic which allows information to be stored with uncertainties, and that is what Mprolog is all about. Mprolog is used to represent a new type of multi-value logic called minimal bounded fuzzy logic.

Prolog is a powerful programming language in which one is able to implement expert systems, but it has no way to handle uncertainties. By restricting values of predicates to either absolute truth or absolute falsehood the use of Prolog as a true expert system tool is severly limited. An example of expert system rules involving a new idea, minimal bounded fuzzy logic, is

$$\text{disease1}\{0.89\} \leftarrow \text{XrayshowsA}\{0.85\}, \text{bloodtestB}\{0.95\}, \text{symptomsC}.$$

$$\text{disease2}\{0.75\} \leftarrow \text{XrayshowsB}\{0.55\}, \text{symptomsB}, \text{symptomsD}.$$

What this means is that a person has disease 1 with a confidence level of 0.89 if the X-ray showed A with 0.85 confidence level, a blood test showed B with 0.95 confidence level and the person is showing symptoms C. By giving each part of the hypothesis a confidence

factor the uncertainty is put directly into the database. It allows you to take into account the fact that several doctors read X-rays differently, and it allows you to take into account the error that a machine might make in performing a blood test. You can have the power of a prolog type language and yet still deal with the uncertainty of nature.

1.2 How to Use Mprolog

Mprolog was written in Common Lisp on the Symbolics 3620. This chapter deals with how to run Mprolog and not the actual implementation of Mprolog. The implementation of Mprolog is described in Chapter 2 and the listing of source code is in Chapter 3.

After logging onto the Symbolics, the source code can be found in the directory "*l:>martin>*". The program consists of three files

PARSER.LISP - The parser for Mprolog

BUILT-IN.LISP - The built in predicate for Mprolog

PROLOG.LISP - The Mprolog interpreter

Each of these files must be compiled and loaded which can be done with either the compile command or from Zmacs. After all the files have been compiled you can simply call Mprolog by invoking the lisp function *mprolog* with no arguments.

Command: (mprolog)

You will now be in Mprolog with the '?-' prompt. To load a file of Mprolog clauses into the database you simply say

?- [file].

This will load the file called *file.mprolog* into the database. To load more than one file they should be seperated by commas as in

?- [file1,file2].

This will load in both *file1.mprolog* and *file2.mprolog*. It will try to load as much as possible even if there are errors in the input. The load takes each clause and *asserts* it into the database. The *assert* will be done as described in the next section.

You may also enter clauses from the prompt with the *assert/retract* commands and later save them with the *tell* command. To save out the database contents you simply say

?- tell(file),listing,told.

which will write the current contents of the database to the output file called *file.mprolog*. The *tell* predicate opens the file for output and all subsequent output is put into the file until a *told* is encountered.

To exit Mprolog you use the built in predicate *quit*. It will ask to make sure you want to quit before leaving.

1.3 Differences From Prolog

The syntax of rules, facts and questions in Mprolog is the same as Prolog except that predicates can now have truth values between 0 and 1 associated with them. For example

1. $Q\{y\}(s_1,\ldots,s_n) :- P1\{x_1\}(t1_1,\ldots,t1_{n_t}),\ldots,PN\{x_n\}(tk_1,\ldots,tk_{n_k})$.
2. $Q\{y\}(s_1,\ldots,s_n)$.
3. ?- $A1\{x_1\}(t1_1,\ldots,t1_{n_t}),\ldots,AN\{x_n\}(tk_1,\ldots,tk_{n_k})$.

Where $y, x_1,\ldots,x_n$ are real number values between 0 and 1 inclusive.

The truth value $\{x_1\}$ can be omitted in which case Mprolog assumes a truth value of one. If all predicates are to have a truth value of one, then the syntax is identical to Prolog.

Mprolog, like Prolog, is structured around a set of built in predicates. With these predicates you are able to add new clauses to the database, retract clauses from the database and change clauses in the database. With rules and facts in the database you can then ask questions. The following is a list of all the built in predicates contained in Mprolog, which is a subset of the built in predicates of standard Prolog. The list contains both differences and likenesses from Prolog.

1. The standard operators *is*, *not*, *!*, =, \=, ==, \==, <, =<, >, >=, *max*, *listing*, *fail*, *true*, *tell*, *told*, *see*, *seen*, *var*, *nonvar*, *atom*, *integer*, *atomic*, *min*, -, *, +, / and *mod* work the same in Mprolog as in Prolog.

2. The predicates **assert, asserta, assertz** are predicates used in Mprolog to add clauses into the database.

   assert($P\{x\}(t_1, \ldots, t_n) :- \alpha$), where $x \in [0, 1]$.

   (a) If the clause is already in the database, with a truth value of y, then the truth value of the clause is changed to $\max(x, y)$.

   (b) If the clause is not in the database it is simply added.

3. **Retract** is a standard Prolog predicate which removes clauses from the database. The syntax and semantics for the retract predicate in Mprolog is

   retract($P\{x\}(t_1, \ldots, t_n) :- \alpha$), where $x \in [0, 1]$.

   (a) If the clause is in the database, with a truth value of y, then the truth value is set to $\min(1 - x, y)$. If $\min(1 - x, y) = 0$ then the clause is removed from the database.

4. **Support** is a new predicate in Mprolog which is not in Prolog. Support is basically like assert except that it allows you to update the truth value of a clause by use of a formula. This allows for the accumulation of evidence for the given clause. The syntax of support has two forms

   (a) support($P\{x\}(t_1, \ldots, t_n) :- \alpha$), where $x \in [0, 1]$.

      i. If the clause is already in the database with a truth value of y, the truth value is replaced with $x + y - x * y$. If this value is 0 then the clause is removed from the database.

      ii. If the clause does not exist in the database then it is added.

   (b) support($(P\{x\}(t_1, \ldots, t_n) :- \alpha), X, Y, f(X, Y)$), where $x \in [0, 1]$.

      i. If the clause is already in the database with a truth value of y, the variable X is instantiated to x and the variable Y is instantiated with y. The truth value of the clause in the database is then replaced with $\min(\max(f(X, Y), 0), 1)$.

If this value is 0 then the clause is removed from the database. The min() and max() functions are used to keep the value between 0 and 1.

ii. If the clause does not exist in the database then it is added.

The action of this predicate is to instantiate X with the truth value of the clause in the support parameter, and Y will be the instantiated with the value of the clause that is in the database. The function f is a mathematical formula used to compute the new truth value based on both X and Y. For example, you could update a clause by

$$\text{support}((P\{0.2\}(a)), X, Y, X * X + Y * Y)$$

- If the clause $P\{0.5\}(a)$ is already in the database then after the support is done $P\{0.29\}(a)$ will be in the the database.

5. **Detract** is a new predicate in Mprolog which is not in Prolog. Detract is basically like support except that the default formula for detraction is different from that for support. Detract is used for the removal of evidence for a given clause. Another difference is that the clause is not added if it is not found in the database. The syntax of detract has two forms

   (a) detract($P\{x\}(t_1, \ldots, t_n) :- \alpha$), where $x \in [0, 1]$.

      i. If the clause is already in the database with a truth value of y, the truth value is replaced with $y - x * y$. If this value is 0 then the clause is removed from the database.

   (b) detract($(P\{x\}(t_1, \ldots, t_n) :- \alpha), X, Y, f(X, Y)$), where $x \in [0, 1]$.

      i. If the clause is already in the database with a truth value of y, the variable X is instantiated to x and the variable Y is instantiated with y. The truth value of the clause in the database is then replaced with $\min(\max(f(X, Y), 0), 1)$. If this value is 0 then the clause is removed from the database. The min() and max() functions are used to keep the value between 0 and 1.

6. To output terms you use either **write** or **writeln**. The predicate *write* takes any number of arguments and outputs them to the current output file. The predicate *writeln* is the same as *write* except that it also outputs a newline when finished.

7. The predicate **quit** is used to exit Mprolog. The predicate does not take any arguments and will ask you to make sure you want to leave Mprolog.

8. The n-ary predicate **lisp** is used to call a lisp function from inside Mprolog. To invoke a lisp function you say **lisp(X,func,a,b,c)** which calls function *func* with parameters *a*, *b* and *c*. The variable *X* is instantiated with the value returned by the function.

9. The built in predicates **read** and **readch** are used to read a string or character respectively. For example, *readch(X)*, will read the next character from the current input and instantiate X with it. The predicate *read(X)* will read in the next string of alphanumeric characters.

10. One of the major drawbacks of Mprolog is that the "|" operator is not implemented. You may use the list notation but can not use "|" to separate the head and tail. This can be accomplished quite easily though, by the use of the *lisp* predicate described above.

Chapter 2

Implementation of Mprolog

2.1 Introduction

The program Mprolog was implemented in Common Lisp on the Symbolics 3620. The program consists of three major parts

Parser – This part performs the input/output for the package. It reads from either files or the terminal and converts clauses into their internal form. It is also able to output a clause based on its internal representation.

Built In Predicates – This contains the wide variety of built in predicates that normal prolog has. These include predicates like *assert*, *retract*, *math operators*, *is* and many others.

Mprolog – This part consists of the actual routines necessary for resolution. This includes functions for renaming clauses, unification, and resolution.

In the following sections a brief description of each part and how it was implemented in lisp will be given.

2.2 A Parser for Mprolog

Mprolog is a simple operator based language since all clauses can be represented by use of operators. The parser used for Mprolog is a stack oriented parser. It separates the input into tokens and then uses the stack oriented parser to change it into an internal form which is more efficient for resolution. The types of tokens that the input is broken into are

1. constant - A constant has three forms

   (a) A prolog variable which starts with an upper case letter or an underline

   (b) A number

   (c) The cut operator which is "!"

2. A delimiter such as (,), {, }, [and]

3. A predicate name which is in reality a postfix operator with 0 or more arguments.

4. An operator, which is basically anything that is not one of the above tokens

All operators, which are also predicates, have the following attributes

precedence - The precedence of the operator. The precedences used in Mprolog are the same ones used in "Programming in Prolog".

class - This tells whether the operator is one of the classes of xfx, xfy, yfx, yfy, fy, fx, xf and yf. These operator classes follow the types used in "Programming in Prolog". For a class of xfy, f is a binary operator with two arguments x and y. The x is used to say any operators in this argument must have strictly lower precedence then operator f. The y represents an equal or lesser precedence.

string name - This is what the actual operator must look like when read in.

print name - What the name looks like when printed. This allows the output to have spaces around it like ' *is* '.

arguments - A list containing the number of arguments that are are allowed. A predicate like *support* can have 1 or 4. Some predicates can have any number of arguments and this is represented with a -1.

function - A lisp function to be called if the predicate is invoked by resolution such as *assert* or *quit*.

After the input is tokenized it is converted into an internal form by use of a stack parser. This will cause a clause like

p(X,Y) :- a(X,b), c(Y,f(d,e)).

to be converted to the internal functional notation of

(bif-implies (p X Y) (bif-and (a X b) (c Y (f d e))))

Notice how everything is either a constant or a predicate. A predicate is the object at the beginning of the list and is a real lisp function or a user defined predicate. During resolution if a built in predicate is to be resolved it is evaluated as a lisp function. This allows resolution to be faster since a table is not searched for built in predicates during resolution but only during input of clauses. There is no need for a search on output since *bif-and* represents both a function and a variable. The variable is equal to the attributes of the operator/function. By looking at the value you can tell the class of operator and also the print name.

2.3 Mprolog's Built in Predicates

In the previous section built in predicates, also called functions, were mentioned. These are the primitives of Prolog which do math operations, input/output and conditionals. All built in predicates are on a list used in the parser. Only the parser uses the list by searching to see if a token is on the list. If it is, then it is a built in predicate and the parser acts accordingly. All built in predicates have one parameter, a list, passed to it when executed. This list contains the arguments to the function and can vary. Each function must check to see whether the arguments are of correct type and if any arguments are of the wrong type, such as a variable that was not instantiated, then an error will be signaled. For example, if the predicate was

(bif-is 1 (bif-plus 2 3))

is executed then it will cause an error since 2, a variable, has not yet been instantiated. The 1 is all right since the *is* predicate is used to instantiate one variable. A built in predicate returns *nil* if the action was unsuccessful. If the predicate was successful then it either returns *t* or a list of variable substitutions to be applied to the goal clause. If in

the above example _2 had been instantiated to 5 then this predicate would have returned (_1(8)). There are a few predicates which are not called as functions and one of which is the cut operator. The cut operator is simply skipped over while resolving predicates and is only used during backtracking. The cut operator does not have any arguments because it justs decides how far to backtrack. The next section describes in more detail how the cut operator works.

2.4 How Mprolog does resolution

The main lisp functions of Mprolog resolution are called *prove*, *unify* and *resolve*. The routine *prove* is called with a goal list. From the goal list an answer list is made which contains all the uninstantiated variables. This answer list is then made the argument of an answer predicate and appended to the end of the goal list. When this predicate is reached the answer is printed out, if it is a new one, and the user is questioned whether he wants to continue. If a *nil* is returned from the answer predicate then another answer is searched for otherwise no more answers will be searched for. The *resolve* routine does the main work of resolution. The *resolve* routine is called with two arguments, the new appended goal list and the length of the list. The length of the list is used in backtracking when the cut operator is used. The action of *resolve* is simple. It takes the head of the goal list and checks first if it is a lisp function which means it is a built in predicate. If it is, the function is called and the return value tells whether it resolved or not. If it is not a built in predicate then it is checked against the database for a predicate that it unifies with. If it unifies, the tail of the unifying clause is appended to the front of the goal list and the substitutes returned from *unify* are applied to the new goal list. The function *resolve* is then called again with this new goal list and new length. If the results from the new call to resolve are *nil* then the next clause in the list is checked to see if it unifies and so on. The function *resolve* returns one of three things.

- The value *nil* which represents failure so the next predicate in the list is tried.
- The value *t* which represents a satisfactory answer was found so exit all recursive calls.

- A number which represents a level to cut back to.

The level is the cutback level. It keeps returning until the length of the goal list is less then this level and then it goes back one more. This will cause it to do the cut as in Prolog.

The Lisp source code for this implementation of Mprolog is given in Chapter 3.

Chapter 3

Mprolog Source Code

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: MPROLOG; Base: 10; Lowercase: Yes; -*-


;;;
;;; PARSER
;;;
;;; This is a stack oriented parser. You call parse-init with a file to be read
;;; or nothing if the terminal is desired. One line from input is read into the
;;; parser at a time. This line is parsed for tokens and they are placed on the
;;; stack and evaluated. The routine parse is called to get one clause from the
;;; input stream. The routine make-clause will convert the print ready form of
;;; the clause into the form stored for the actual use by prolog.


;;;
;;; Global variables:
;;;
;;; input-char - holds the current input character
;;; input-eof - if the current input stream is at the end of file
;;; input-errcol - the column where the error occurred at when reading
;;; input-length - the length of the current line of input
;;; input-line - the current line number
;;; input-position - the current position in the input-line
;;; input-stream - where the input is comming from, either the terminal or a file
;;; output-stream - where the output is going, changed by prediactes tell and told
;;; input-string - string holding the current line of input
;;; input-token - the current token that was input
;;;


(defvar input-char " ")
(defvar input-eof nil)
(defvar input-errcol 0)
(defvar input-error nil)
(defvar input-length 0)
```

```
(defvar input-line 0)
(defvar input-position 0)
(defvar input-stream *terminal-io*)
(defvar output-stream *terminal-io*)
(defvar input-string " ")
(defvar input-token nil)


;;;
;;; Function : collapse
;;;
;;; Input : name - the name of the function to collapse
;;; x - the atom that is being added to the list
;;; y - the list which may already be made up of name's
;;; Output : list - containing the new combined expression
;;;
;;; This routine simply collapes "," and ";" into one "," or ";" respectively.
;;; It will take ("," a ("," b c)) and make ("," a b c). The same is done for
;;; a group of ";"'s.
;;;


(defun collapse (name x y)
  (cons name (if (and (listp y) (equal (car y) name))
                 (if (and (listp x) (equal (car x) name))
                     (append (cdr x) (cdr y))
                     (cons x (cdr y)))
                 (if (and (listp x) (equal (car x) name))
                     (append (cdr x) (list y))
                     (list x y)))))


;;;
;;; Function : get-char
;;;
;;; Input : none
;;; Output : character
;;;
;;; Reads one character from the current input string. If it is at the end of the
;;; string then a knew string is read in. If an EOF occurs on input then the character
;;; # \ End is returned. It also returns an extra space at the end of each line
;;; of input. The position pointer is incremented by one when leaving to point to the
;;; next character to be read.
;;;


(defun get-char ()
  (setq input-char (loop while (not input-eof)
                         if(= input-length input-position)
```

```
                    do (get-line)
                   else if(equal input-length input-position)
                        return # \ Space
                   else
                    return (char input-string input-position)
                   finally (return # \ End)))
  (incf input-position)
  input-char)


;;;
;;; Function : get-expression
;;;
;;; Input : none
;;; Output : list - containing an expression
;;;
;;; This will return an expression read in from input set up by parse-init. If an
;;; error occurs then input-error holds a string describing the error. This routine
;;; is called when a left paren occurs in get-predicate. It is just a stack oriented
;;; parser with the precedences of operators already defined in the bif-list (built
;;; in function list).
;;;


(defun get-expression ()
  (loop with operator-stack = '((end-of-stack 1000)) and
        operand-stack = nil and
        operator-next = '(operator del-start del-stop) and
        operand-next    '(funcname constant del-start) and
        expected = nil
       initially (get-token) (setq expected operand-next)
       while (and (not input-eof) (not input-error) (member (car input-token) expected)
                  (or (cdr operator-stack) (not (equal (cadr input-token) 'bif-dot))))
; do (print (list "start loop" input-token))
  do (case (get-operation operator-stack)
       (0 (push (cadr input-token) operand-stack) (setq expected operator-next)
          (get-token))
       (1 (push (get-predicate (cadr input-token) (cdr input-token)) operand-stack)
          (setq expected operator-next))
       (2 (let* ((z (pop operator-stack)) (y (pop operand-stack))
                 (x (if (member z '(fx fy)) nil (pop operand-stack))))
            (if x (if (or (equal (car z) 'bif-and) (equal (car z) 'bif-or))
                      (push (collapse (car z) x y) operand-stack)
                      (push (list (car z) x y) operand-stack))
                  (push (list (car z) y) operand-stack))))
       (3 (push (cdr input-token) operator-stack) (setq expected operand-next)
          (get-token))
       (4 (push (cdr input-token) operator-stack) (get-token))
```

```
        (5 (pop operator-stack) (get-token))
        (6 (return (pop operand-stack)))
        (7 (push (list (cadr input-token) (pop operand-stack)) operand-stack)
           (get-token)))
; do (print (list "endloop" operator-stack "operands" operand-stack))
  finally (if input-error (return nil)
              (if (member (car input-token) expected)
                  (return (pop operand-stack))
                  (if (equal expected operator-next)
                      (return (make-error "Expected an operator"))
                      (return (make-error "Expected an operand")))))))


;;;
;;; Function : get-line
;;;
;;; Input : none
;;; Output : none
;;;
;;; This will read one line of input from the current input stream into 'input-string'.
;;; It resets the position and increments the number of lines read. Upon reaching
;;; EOF it sets 'input-eof' to true. It sets the 'input-length' to the length of the
;;; string read and also sets the current 'input-char' to a space. This will end the
;;; current token and force a get-char to occur.
;;;


(defun get-line ()
  (setq input-position 0)
  (setq input-string (read-line-trim input-stream nil # \ End nil))
  (cond ((equal # \ End input-string) (setq input-eof t))
        (t (incf input-line)
           (setq input-string (concatenate 'string input-string " "))
           (setq input-length (string-length input-string))
           (setq input-char # \ Space))))


;;;
;;; Function : get-number
;;;
;;; Input : none
;;; Output : list - of the form (number number-read)
;;;
;;; This reads a number from the current input stream. A number must start with a
;;; digit so that .12 is illegal but 0.12 is ok. If a number is to end the clause
;;; then it must not end with a period. This is because the period in 1. will be
;;; taken as a real number and not as 1 followed by the end of clause period.
;;;
```

```
(defun get-number ()
  (list 'constant (loop with num = nil and okperiod = t
                        while (or (is-numeric input-char)
                                  (and okperiod (equal input-char # \ .)))
                        do (setq num (concatenate 'string num (string input-char)))
                        if(equal input-char # \ .)
                         do (setq okperiod nil)
                            (if (is-numeric (char input-string input-position)) (get-char))
                        else do (get-char)
                        finally (return (read (make-string-input-stream num))))))


;;;
;;; Function : get-operation
;;;
;;; Input : top - operator stack
;;; Output : number - operation to do with the stacks and the current token
;;;
;;; This routine will look at the current token and the top of the operator stack
;;; and decide what operation is to be performed on it. Operations are the following:
;;; 0 - push token onto operand stack, set expected token to be an operator,
;;; get the next token
;;; 1 - get parameters of token and push onto stack, set expected token to be an
;;; operator, get the next token
;;; 2 - the token is an operator so build the function and its parameters
;;; 3 - push the token on the operator stack, set expected token to be an operand,
;;; get the next token
;;; 4 - push the token on the operator stack, get the next token
;;; 5 - pop the operator stack, get the next token
;;; 6 - return the value off the top of the operand stack
;;; 7 - build a postfix expression and push onto operand stack, get the next token
;;; 8 - do nothing because an error occurred
;;;


(defun get-operation (operator-stack)
  (let ((token (car input-token))
        (name (cadr input-token))
        (prec (caddr input-token))
        (class (cadddr input-token)))
    (cond ((equal token 'constant) 0)
          ((equal token 'funcname) 1)
          ((equal name 'open-paren) 4)
          ((equal name 'close-bracket)
           (if (equal (caar operator-stack) 'end-of-stack) 6 2))
          ((equal name 'close-paren)
           (if (equal (caar operator-stack) 'open-paren) 5
               (if (equal (caar operator-stack) 'end-of-stack) 6 2)))
```

```
        ((< prec (cadar operator-stack))
         (if (member class '(xf yf)) 7 3))
        ((> prec (cadar operator-stack)) 2)
        ((member class '(yf yfx yfy)) 2)
        ((member (caddar operator-stack) '(xfy yfy fy))
         (if (member class '(xf yf)) 7 3))
        (t (make-error "Illegal operator use") 8))))


;;;
;;; Function : get-operator
;;;
;;; Input : none
;;; Output : list - of the form (operator (name prec class numparams))
;;;
;;; This reads an operator from the current input stream. An operator is anything
;;; that does not contain letters, numbers or delimiters. An error is signaled when
;;; the operator has not been declared as such. If the operator is made up of letters
;;; then it will be caught in the get-symbol routine.
;;;


(defun get-operator ()
  (loop with op = (string input-char)
        initially (get-char)
        while (not (or (is-white input-char) (is-delimiter input-char)
                       (alphanumericp input-char) (equal input-char # \  )))
        do (setq op (concatenate 'string op (string input-char))) (get-char)
        finally (return (let ((x (cadr (member op bif-list :test 'equal))))
                          (cons (if x (if (member (caddr x) '(fx fy)) 'funcname 'operator)
                                    (and (make-error (concatenate 'string
                                                                  "Unknown operator : " op))
                                         'operator))
                                x)))))


;;;
;;; Function : get-predicate
;;;
;;; Input : head - name of the predicate read in
;;; built - if the head is also a built in function then this is a ist of
;;; attributes
;;; Output : list - representing the predicate
;;;
;;; Returns the predicate read in a lisp form as follows:
;;; test(a,b,c,...,z) -> (test a b c ... z)
;;; test0.12(a,b,c) -> (bif-value 0.12 test a b c)
;;; test -> test
;;; test0.12 -> (bif-value 0.12 test)
```

```
;;;
;;; If the head is a built in function then the actual builtin function name is supplied.
;;;


(defun get-predicate (head built)
  (if (equal head 'bif-load)
      (let ((body (get-expression)))
        (cond ((equal (cadr input-token) 'close-bracket)
               (get-token) (if (and (listp body) (equal (car body) 'bif-and))
                               (cons head (cdr body)) (list head body)))
              ((make-error "Expected a closing bracket"))))
      (let ((value (if (equal (cadr (get-token)) 'open-curly)
                       (let ((x (get-token)))
                         (cond ((equal (cadr x) 'close-curly) (get-token) 1)
                               ((and (equal (car x) 'constant) (numberp (cadr x))
                                     (equal (cadr (get-token)) 'close-curly))
                                (get-token) (cadr x))
                               ((make-error "Expected a closing curly bracket") 1)))
                       1))
            (body (if (and (equal (cadr input-token) 'open-paren) (not input-error))
                      (let ((x (get-expression)))
                        (if (equal (cadr input-token) 'close-paren)
                            (and (get-token) (if (and (listp x) (equal (car x) 'bif-and))
                                                 (cdr x) (if x (list x) x)))
                            (make-error "Expected a closing parentheis")))))
            (builtin (cond ((cdr built) built)
                           (t (cadr (if (equal (char head 0) # \ ')
                                        (member (substring head 1 (- (string-length head) 1))
                                                bif-list :test 'equal)
                                        (member head bif-list :test 'equal)))))))
        (if (and builtin (functionp head))
            (setq head (concatenate 'string "" (car (eval (car builtin))) "")))
        (if (and builtin (not (eval (append (list '/= -1 (length body)) (cdddr builtin)))))
            (cons (car builtin) body)
            (if body (append (if (< value 1) (list 'bif-value value head) (list head)) body)
                (if (< value 1) (list 'bif-value value head) head))))))


;;;
;;; Function : get-string
;;;
;;; Input : none
;;; Output : list - of the form (funcname string-read)
;;;
;;; Reads a string from the current input stream inclosed between '.
;;;
```

```
(defun get-string (sdel)
  (loop with str = nil
        initially (get-char)
        do (setq str (concatenate 'string str (string input-char))) (get-char)
        until (or input-eof (equal sdel input-char))
        finally (if input-eof (make-error "Unexpected end of file")
                    (if (equal sdel # \ ') (setq str (concatenate 'string (string # \ ')
                                                                  str (string # \ '))))))
                (setq input-char # \ Space) (return (list 'funcname str))))


;;;
;;; Function : get-symbol
;;;
;;; Input : stype - either 'constant or 'funcname
;;; Output : list - of the form (stype string-read)
;;;
;;; Reads the next symbol from the current input stream. A variable starts with an
;;; upper case letter or underline. If it is a variable then it is interned into
;;; the symbol table and the name rather then the string is returned. If the name is
;;; an operator then (operator (name prec class ...)) is returned instead.
;;;


(defun get-symbol (stype)
  (let ((x (loop with str = nil
                 while (or (is-alpha input-char) (is-numeric input-char))
                 do (setq str (concatenate 'string str (string input-char))) (get-char)
                 finally (if (equal stype 'funcname)
                             (return (cond ((cadr (member str bif-list :test 'equal)))
                                           ((list str))))
                             (return (list (if (equal str "_") (make-var) (intern str))))))))
    (cons (if (and (> (length x) 1) (not (equal (caddr x) 'fx)))
              'operator stype) x)))


;;;
;;; Function : get-token
;;;
;;; Input : none
;;; Output : list - a description of the next token found in the input stream.
;;;
;;; The token will be one of the following:
;;; (constant interned-name)
;;; (constant (bif-cut))
;;; (constant number)
;;; (delimiter name)
;;; (end-of-file)
;;; (funcname string-name)
```

```
;;; (funcname memlist)
;;; (operator (name prec class ...))
;;;


(defun get-token ()
  (get-white)
  (setq input-errcol (- input-position 1))
  (setq input-token
        (cond ((equal input-char # \ !) (get-char) (list 'constant '(bif-cut)))
              ((member input-char '(# \ ' # \ ")) (get-string input-char))
              ((is-upper input-char) (get-symbol 'constant))
              ((is-alpha input-char) (get-symbol 'funcname))
              ((equal input-char # \ .) (if (is-numeric (char input-string input-position))
                                            (get-number) (get-operator)))
              ((is-numeric input-char) (get-number))
              ((is-delimiter input-char) (let ((x input-char)) (get-char) (is-delimiter x)))
              ((get-operator)))))


;;;
;;; Function : get-white
;;;
;;; Input : none
;;; Output : none
;;;
;;; Skips over all white spaces in the current input stream. A white space is either tabs
;;; or spaces.
;;;


(defun get-white ()
  (loop while (and (is-white input-char) (not input-eof))
        do (get-char)))


;;;
;;; Function : is-alpha
;;; is-delimiter
;;; is-numeric
;;; is-upper
;;; is-white
;;;
;;; Input : ch - character
;;; Output : nil or t
;;;
;;; Determines whether the character is in the desired character class. In the case of
;;; is-delimiter a list of the form (delimiter name) is returned if it is in fact a delimiter.
;;;
```

```
(defun is-alpha (ch)
  (or (equal ch # \ _ ) (alpha-char-p ch)))


(defun is-delimiter (ch)
  (cadr (member ch '(# \ ( (del-start open-paren 300) # \ ) (del-stop close-paren)
                     # \ [ (funcname bif-load) # \ ] (del-stop close-bracket)
                     # \  (del-cstart open-curly) # \  (del-cstop close-curly)
                     # \ End (end-of-file) # \ . # \ ! # \ " # \ ' t))))


(defun is-numeric (ch)
  (and (char> = ch # \ 0) (char< = ch # \ 9)))


(defun is-upper (ch)
  (or (upper-case-p ch) (equal ch # \ _ )))


(defun is-white (ch)
  (or (equal ch # \ Space) (equal ch # \ Tab)))


;;;
;;; Function : make-clause
;;;
;;; Input : x - clause read by parse to be converted
;;; Output : list - contains clause form used by theorem prover
;;;
;;; This takes a print ready form of a clause and returns the following:
;;; (bif-implies head tail) -> (head tail (length of tail) clause)
;;; head -> (head nil 0 clause)
;;; The function parms is used to remove any unwanted bif-and in the start of head or tail.
;;;


(defun make-clause (x)
  (let* ((implies (and (listp x) (equal (car x) 'bif-implies)))
         (head (if implies (cadr x) x))
         (tail (parms (if implies (caddr x)))))
    (list head tail (length tail) x)))


;;;
;;; Function : make-error
;;;
;;; Input : err - string describing the error
;;; Output : none
;;;
```

```
(defun make-error (err)
  (setq input-error (concatenate 'string
                                 (substring input-string 0 input-errcol)
                                 " <- " err (if (equal *terminal-io* input-stream) ""
                                                (concatenate 'string " ,line "
                                                             (write-to-string input-line)))
                                 " -> "
                                 (substring input-string input-errcol))))


;;;
;;; Function : parms
;;;
;;; Input : x - a list or an atom
;;; Output : list - with bif-and removed
;;;
;;; If the list passed is of the form (bif-and a b c) then (a b c) is returned.
;;; If the form of the list passed is (a b c) then ((a b c)) is returned.
;;;


(defun parms (x)
  (if (and (listp x) (equal (car x) 'bif-and)) (cdr x) (if x (list x))))


;;;
;;; Function : parse
;;;
;;; Input : none
;;; Output : none
;;;
;;; Reads in one clause from the input stream set up by parse-init.
;;;


(defun parse ()
  (setq input-error nil)
  (get-char) (get-white)
  (if input-eof nil
      (let ((x (get-expression)))
        (if (not (or input-error (equal (cadr input-token) 'bif-dot)))
            (make-error "Unexpected end of clause"))
        (cond (input-error (loop while (not (or input-eof
                                                (equal (cadr input-token) 'bif-dot)))
                                 do (get-token))
                           (princ input-error output-stream) (list nil))
              (t x)))))
```

```
            (if (> (length body) 3)
                (print-clause (cons (car body) (cddr body)) nil prec)
                (print-clause (caddr body) nil prec))))
        (if (or parens (> prec inprec)) (princ ")" output-stream))))
    (t (if parens (princ "(" output-stream))
        (princ clause output-stream) (if parens (princ ")" output-stream))))))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: MPROLOG; Base: 10; Lowercase: Yes -*-


;;;
;;; BUILT IN MPROLOG FUNCTIONS
;;;


;;;
;;; Global variables:
;;;
;;; bif-list - this holds all built in functions as (name1 (func prec class nargs)
;;; name2 ...)
;;;


(defvar bif-list nil)


;;;
;;; Function : make-op
;;;
;;; Input : prec - precedence of the operator
;;; spec - class of operator, one of xfx, xfy, yfx, yfy, fx
;;; name - string name of the function
;;; space - whether to put spaces around the actual print name
;;; func - the actual lisp function that will be called
;;; args - number of arguments the function can take, -1 means any number
;;; Output : none
;;;
;;; This adds a function to the function list. If there is no args parameter then the
;;; number of arguments is taken to be the length of the class - 1. If func is not a
;;; string then it creates a parameter called 'func and gives it a value equal to the
;;; print name of the function.
;;;


(defun make-op (prec spec name space func & optional (args nil))
  (push (append (list func prec spec)
                (if args (if (listp args) args (list args))
                  (list (- (string-length spec) 1)))) bif-list)
  (push name bif-list)
  (if (not (stringp func))
    (eval (defparameter ,func '(        ,(if space (concatenate 'string " " name " ") name)
                                ,(intern (string-upcase spec)) ,prec)))))


;;;
;;; Function : bif-name
```

```
;;;
;;; Input : pname - list which holds a function type name
;;; Output : string - the actual print name of pname
;;;
;;; This takes any list containing a function and returns the print name of that function.
;;; A mprolog function that is.
;;;


(defun bif-name (pname)
  (cond ((equal (car pname) 'bif-value)
         (list (concatenate 'string (caddr pname) "" (write-to-string (cadr pname)) "")
               'fx 0))
        ((equal (car pname) 'bif-load) (list 'bif-load 'fx 0))
        ((functionp (car pname)) (eval (car pname)))
        ((let ((x (cadr (member (car pname) bif-list :test 'equal))))
           (if x (list (car pname) (caddr x) (cadddr x)) nil)))
        (t (list (car pname) 'fx 0))))


;;;
;;; The next group of function calls set up the built in functions of mprolog.
;;;


(setq bif-list nil)
(make-op 255 'xf "." nil 'bif-dot)
(make-op 254 'xfx ":-" t 'bif-implies)
(make-op 254 'fx "?-" nil 'bif-question)
(make-op 253 'xfy ";" nil 'bif-or)
(make-op 252 'xfy "," nil 'bif-and)
(make-op 250 'fx "spy" nil 'bif-spy)
(make-op 250 'fx "nospy" nil 'bif-nospy)
(make-op 60 'fx "not" nil 'bif-not)
(make-op 40 'xfx "is" t 'bif-is)
(make-op 40 'xfx "=.." t 'bif-univ)
(make-op 40 'xfx "=" t 'bif-equality)
(make-op 40 'xfx "\=" t 'bif-non-equality)
(make-op 40 'xfx "<" t 'bif-less-than)
(make-op 40 'xfx "=<" t 'bif-less-than-or-equal)
(make-op 40 'xfx ">=" t 'bif-greater-than-or-equal)
(make-op 40 'xfx ">" t 'bif-greater-than)
(make-op 40 'xfx "==" t 'bif-strict-equal)
(make-op 40 'xfx "\==" t 'bif-not-strict-equal)
(make-op 35 'fx "max" t 'bif-max 2)
(make-op 35 'fx "min" t 'bif-min 2)
(make-op 31 'yfx "-" nil 'bif-minus)
(make-op 31 'yfx "+" nil 'bif-plus)
(make-op 21 'yfx "/" nil 'bif-divide)
```

```
(make-op 21 'yfx "*" nil 'bif-times)
(make-op 11 'xfx "mod" t 'bif-mod)


(make-op 0 'fx "write" nil 'bif-write -1)
(make-op 0 'fx "writeln" nil 'bif-writeln -1)
(make-op 0 'fx "assert" nil 'bif-assertz 1)
(make-op 0 'fx "asserta" nil 'bif-asserta 1)
(make-op 0 'fx "assertz" nil 'bif-assertz 1)
(make-op 0 'fx "retract" nil 'bif-retract 1)
(make-op 0 'fx "support" nil 'bif-support '(1 4))
(make-op 0 'fx "detract" nil 'bif-detract '(1 4))
(make-op 0 'fx "listing" nil 'bif-listing -1)
(make-op 0 'fx "!" nil 'bif-cut 0)
(make-op 0 'fx "fail" nil 'bif-fail 0)
(make-op 0 'fx "true" nil 'bif-true 0)
(make-op 0 'fx "quit" nil 'bif-quit 0)
(make-op 0 'fx "tell" nil 'bif-tell 1)
(make-op 0 'fx "told" nil 'bif-told 0)
(make-op 0 'fx "clear" nil 'bif-clear 0)
(make-op 0 'fx "lisp" nil 'bif-lisp -1)


;;;
;;; These are the actual functions that will be called when trying to resolve a question.
;;; The first few are dummy functions and will never be called.
;;;


(defun bif-cut (args) (bif-error "Undefined function cut:" args))
(defun bif-dot (args) (bif-error "Undefined function dot:" args))
(defun bif-implies (args) (bif-error "Undefined function implies:" args))
(defun bif-question (args) (bif-error "Undefined function question:" args))
(defun bif-or (args) (bif-error "Undefined function or:" args))
(defun bif-and (args) (bif-error "Undefined function and:" args))
(defun bif-value (args) (bif-error "Undefined function value:" args))
(defun bif-spy (args) (bif-error "Undefined function spy:" args))
(defun bif-nospy (args) (bif-error "Undefined function nopsy:" args))


(defun bif-fail (args) nil)
(defun bif-true (args) t)
(defun bif-not (args) (not (resolve (rename args) 1)))


(defun bif-tell (args)
  (cond ((and (listp args) (stringp (car args)))
         (if (not (equal output-stream *terminal-io*)) (close output-stream))
         (cond ((setq output-stream (open (concatenate 'string (car args) ".mprolog")
```

```
                                        :direction :output)) t)
                (t (setq output-stream *terminal-io*) nil)))))


(defun bif-told (args)
  (if (not (equal output-stream *terminal-io*)) (close output-stream))
  (setq output-stream *terminal-io*) t)


(defun bif-lisp (args)
  (if (and (listp args) (stringp (car args)))
     (let ((funcname (intern-soft (string-upcase (car args)))))
      (if funcname (apply funcname (cdr args))))))


(defun bif-support (clause & optional (top nil) (support t))
  (setq syms 0)
  (let* ((var1 (if (cadr clause) (cadr clause) 'X))
        (var2 (if (cadr clause) (caddr clause) 'Y))
        (formula (if (cadr clause) (cadddr clause)
                     '(bif-minus (bif-plus X Y) (bif-times X Y))))
        (repl (if (is-var var1)
                  (if (is-var var2)
                     (let* ((c (make-clause (rename (car clause))))
                            (x (clause-value (car c))))
                       (if (is-var (car c))
                          (bif-error "Head of clause is a variable: " (car c))
                          (if (functionp (cadr x))
                             (bif-error "Trying to redefine system predicate")
                             (if (not (numberp (car x)))
                                (bif-error "Predicate value is not a number: " (car x))
                                (add-clause x c var1 var2 formula top support)))))
                     (bif-error "Third argument to support should be a variable: " var2))
                  (bif-error "Second argument to support should be a variable: " var1))))
    (cond (repl (set-make-var) (setq syms maxsyms) repl)
          (t (setq syms maxsyms) nil))))


(defun bif-asserta (clause) (bif-support (append clause '(X Y (bif-max X Y))) t))
(defun bif-assertz (clause) (bif-support (append clause '(X Y (bif-max X Y))) nil))
(defun bif-retract (clause) (bif-support (append clause '(X Y (bif-min (bif-minus 1 X) Y)))
                                         nil nil))
(defun bif-detract (clause) (bif-support (append clause '(X Y (bif-minus X (bif-times X Y))))
                                         nil nil))


(defun bif-listing (clause)
  (loop with ret = nil and prev = nil
        for x in hypothesis
```

```
        if(and (listp x) (or (not (car clause)) (equal (car clause) prev)))
          do (loop for y in x
                   do (print-clause (cadddr y)) (princ "." output-stream)
                                                (terpri output-stream) (setq ret t))
        do (setq prev x)
        finally (return ret)))


(defun bif-is (args)
  (if (is-var (car args))
      (let ((is (bif-eval (cadr args)))) (if is (list (car args) (list is))))
      (bif-error "First argument to IS is not a variable: " (car args))))


(defun bif-univ ())
(defun bif-equality (args) (unify (car args) (cadr args)))
(defun bif-non-equality (args) (not (bif-equality args)))
(defun bif-strict-equal (args) (equal (car args) (cadr args)))
(defun bif-not-strict-equal (args) (not (bif-strict-equal args)))


(defun bif-minus (a) (math-operation '- a))
(defun bif-plus (a) (math-operation '+ a))
(defun bif-divide (a) (math-operation '/ a))
(defun bif-times (a) (math-operation '* a))
(defun bif-mod (a) (math-operation 'mod a))
(defun bif-less-than (a) (math-operation '< a))
(defun bif-less-than-or-equal (a) (math-operation '< = a))
(defun bif-greater-than-or-equal (a) (math-operation '> = a))
(defun bif-greater-than (a) (math-operation '> a))
(defun bif-max (a) (math-operation 'max a))
(defun bif-min (a) (math-operation 'min a))


;;;


(defun bif-write (a) (loop for x in a do (print-clause x) (princ " " output-stream)) t)
(defun bif-writeln (a) (bif-write a) (terpri output-stream) t)
(defun bif-answer (args)
  (if args
      (loop with new = nil
            initially (let ((tmp syms))
                        (setq syms 0) (setq new (rename args)) (setq syms tmp))
            for x in answers
            if (equal new x) return nil
            finally (push new answers)
                    (return (loop initially (terpri output stream)
                                  for y in new
```

```
                    do (terpri output-stream) (princ (car y) output-stream)
                            (princ " = " output-stream)
                                    (print-clause (cadr y))
                                    (princ " " output-stream)
                    finally (return (not (equal (read-char *terminal-io*)
                                            # \ ;))))))) t))


(defun bif-clear (args)
  (terpri output-stream) (princ "Are you sure you want to clear database ? " output-stream)
  (cond ((member (read-char *terminal-io*) '(# \ y # \ Y)) (setq hypothesis nil) t)))


(defun bif-quit (args)
  (terpri output-stream) (princ "Are you sure you want to quit ? " output-stream)
  (if (member (read-char *terminal-io*) '(# \ y # \ Y))
     (or (terpri output-stream) (terpri output-stream)
        (throw 'exit-mprolog "Exit Mprolog")) (terpri output-stream)) nil)


(defun bif-load (args)
  (loop with ret = nil
        for x in args
        if(stringp x)
         if(parse-init x)
           do (loop with eof = nil
                    do (if (setq eof (parse))
                           (if (or (not (listp eof)) (car eof)) (bif-assertz (list eof))))
                    until (not eof))
              (setq ret '(nil))
        else
          do (bif-error "Could not open file: " x)
        else
          do (bif-error "Illegal file name: " x)
        finally (parse-init nil) (return ret)))
;;;


(defun bif-error (s & optional (e "")) (princ s output-stream) (print-clause e)
       (terpri output-stream) nil)


(defun bif-eval (f)
  (if (and f (listp f))
     (if (functionp (car f)) (funcall (car f) (cdr f))
        (bif-error "Undefined function call: " (car f)))
     (if (numberp f) f (bif-error "Argument to math operation was not a number: " f))))
```

```
(defun math-operation (op args)
  (let* ((x (bif-eval (car args))) (y (if x (bif-eval (cadr args)))))
    (if (and (numberp x) (numberp y))
        (if (and (equal op '/) (equal y 0))
            (bif-error "Divide by zero error")
            (funcall op x y)))))


(defun add-clause (name clause var1 var2 formula top support)
  (loop with hyp = nil and last = nil and repl with nil
        for pred in hypothesis
        if(and (not repl) (equal last (cadr name)))
          collect (loop with tmp = nil and val = nil
                        for x in pred
                        do (setq tmp (clause-value (car x)))
                        if(and (not repl) (equal (caddr name) (caddr tmp))
                               (equal (cadr x) (cadr clause)))
                          appending
                            (let ((new (bif-eval
                                         (sub formula
                                              (setq repl (list
                                                           var1 (list (car name))
                                                           var2 (list (car tmp))))))))
                              (if (numberp new) (setq new (max (min new 1) 0)))
                              (if (numberp new)
                                  (if (equal new 0) nil
                                      (list (if (not (equal new (car name)))
                                                (replace-value (max (min new 1) 0)
                                                               (cadddr clause)) clause)))
                                  (and (setq repl t) (list x)))) into val
                        else
                          appending (list x) into val
                        finally (return (if repl val
                                            (if support
                                                (and (setq repl '(nil))
                                                     (if top (append (list clause) pred)
                                                         (append pred (list clause))))
                                                (and (setq repl nil) val)))))
            into hyp
        else
          collect pred into hyp
        do (setq last pred)
        finally (if repl (if (listp repl) (setq hypothesis hyp) (setq repl nil))
                     (if support (and (setq repl '(nil))
                                      (setq hypothesis (append hyp (list (cadr name)
                                                                         (list clause)))))
                         (setq repl (bif-error "Could not find clause to detract from"))))
                (return repl)))
```

```
(defun replace-value (new cl)
  (let* ((implies (and (listp cl) (equal (car cl) 'bif-implies)))
         (head (if implies (cadr cl) cl))
         (tail (caddr cl)))
    (setq head (if (equal new 1)
                   (if (listp head) (if (equal (car head) 'bif-value)
                                        (if (cdddr head) (cddr head) (caddr head)) head) head)
                   (if (listp head) (if (equal (car head) 'bif-value)
                                        (cons 'bif-value (cons new (cddr head)))
                                        (cons 'bif-value (cons new head)))
                       (list 'bif-value new head))))
    (make-clause (if implies (list 'bif-implies head tail) head))))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: MPROLOG; Base: 10; Lowercase: Yes -*-


;;;
;;; MPROLOG
;;;


;;;
;;; Global variables:
;;;
;;; syms - current symbol number to be created by make-var
;;; maxsyms - maximum number of syms to be created in any one clause
;;; hypothesis - the current list of clauses entered
;;; answers - the current list of answers that have been proven
;;;


(defvar syms 0)
(defvar maxsyms 0)
(defvar hypothesis nil)
(defvar answers nil)


;;;
;;; Function : answer-list
;;;
;;; Input : c - a question clause
;;; Output : list - containing variables and thier print names.
;;;
;;; Goes through the current list getting the variables into a list that looks
;;; like (("pname1" var1) ("pname2" var2) ...). This will hold the answer obtained
;;; from the resolution.
;;;


(defun answer-list (c & optional ans)
  (if c (if (listp c)
            (answer-list (cdr c) (answer-list (car c) ans))
            (if (is-var c) (append ans
                                   (loop for x in ans
                                         do (if (equal c (cadr x)) (return nil))
                                         finally (return (list (list (symbol-name c) c)))))
                ans)) ans))


;;;
;;; Function : clause-value
;;;
```

```
;;; Input : c - predicate
;;; Output : list - holds the value and actual predicate with value removed
;;;
;;; This routine takes a predicate with optional value part and returns with
;;; a list containing the value and predicate with optional vaue removed.
;;;


(defun clause-value (c)
  (if (and (listp c) (equal (car c) 'bif-value))
      (list (cadr c) (caddr c) (if (cdddr c) (cddr c) (caddr c)))
      (list 1 (if (listp c) (car c) c) c)))


;;;
;;; Function : cutit
;;;
;;; Input : x - last resovant value returned
;;; len - length of current goal
;;;
;;; If x = nil then return nil. If x = t then return t. If x is neither one then
;;; it must be an integer. If the integer is less than or equal to the length then
;;; it should still be cut otherwise the cut should stop at this point.
;;;


(defun cutit (x len) (if (numberp x) (if (< x 0) nil (if (> len x) x -1)) x))


;;;
;;; Function : is-var
;;;
;;; Input : v - a possible variable
;;; Output : nil or t
;;;
;;; Checks to see whether v is a variable or not. A variable is a symbol but is not
;;; a function. Builtin functions such as bif-or are both.
;;;


(defun is-var (v) (and v (symbolp v) (not (functionp v))))


;;;
;;; Function : make-var
;;;
;;; Input : none
;;; Output : none
;;;
```

```
;;; Generates a new variable starting with "_ " and interns it.
;;;

(defun make-var () (intern (concatenate 'string "  " (write-to-string (incf syms))) 'mprolog))

(defun pred-name (x) (if (listp x) (if (equal (car x) 'bif-value) (caddr x) (car x)) x))

;;;
;;; Function : mprolog
;;;
;;; Input : none
;;; Output : none
;;;
;;; The mprolog loop. Gets a question and calls prove over and over again.
;;;

(defun mprolog ()
  (setq output-stream *terminal-io*)
  (princ (catch 'exit-mprolog
           (setq hypothesis nil)
           (setq syms (setq maxsyms 0))
           (loop while t
                 do (terpri output-stream)
                    (princ "?- " output-stream)
                    (parse-init nil)
                    (let ((x (parse))) (if (and x (or (not (listp x)) (car x)))
                                           (prove (if (and (listp x) (equal 'bif-and (car x)))
                                                      (cdr x) (list x))))))) nil)

;;;
;;; Function : prove
;;;
;;; Input : conclusion - holds the clause you want to prove
;;; Output : none
;;;
;;; Proves the conclusion on the current hypothesis.
;;;

(defun prove (conclusion)
  (setq syms maxsyms)
  (setq answers nil)
  (let ((x (resolve (rename (append conclusion (list (cons 'bif-answer
                                                           (answer-list conclusion))))))
```

```
                (+ (length conclusion) 1))))
    (terpri output-stream)
    (princ (if (equal x t) "yes" "no") output-stream))
  (terpri output-stream))


;;;
;;; Function : unify
;;;
;;; Input : p1,p2 - these are two predicates to be unified.
;;; Output : list - containing the most general unifier (MGU)
;;;


(defun unify (p1 p2) (unifier p1 p2 nil))


;;;
;;; Function : unifier
;;;
;;; Input : c1,c2 - the two items to be unified
;;; mgu - the current mgu obtained so far
;;; Output : list - nil if it did not unify otherwise the substitutions
;;;
;;; If the two items unify but without a substitution then a (nil) is returned.
;;; The substitution on the two clauses is not done until it is needed.
;;;


(defun unifier (c1 c2 mgu)
  (cond ((is-var c1)
         (if (equal c1 c2)
            (or mgu '(nil))
            (let ((x (member c1 mgu)))
              (if x (unifier (caadr x) c2 mgu)
                 (append (list c1 (sub (list c2) mgu)) (sub mgu (list c1 (list c2))))))))
        ((is-var c2)
         (if (equal c1 c2)
            (or mgu '(nil))
            (let ((x (member c2 mgu)))
              (if x (unifier c1 (caadr x) mgu)
                 (append (list c2 (sub (list c1) mgu)) (sub mgu (list c2 (list c1))))))))
        ((atom c1) (if (if (atom c2) (equal c1 c2)
                           (if (and (equal (car c2) 'bif-value) (not (cdddr c2)))
                              (equal (caddr c2) c1))) (or mgu '(nil))))
        ((atom c2) nil)
        ((equal (car c1) 'bif-value)
         (if (> = (cadr c1) (if (equal (car c2) 'bif-value) (cadr c2) 1))
            (unifier (cddr c1) (cddr c2) mgu)))
```

```
        ((equal (car c2) 'bif-value) (unifier c1 (if (cdddr c2) (cddr c2) (caddr c2)) mgu))
        ((let ((x (unifier (car c1) (car c2) mgu)))
           (if x (unifier (cdr c1) (cdr c2) x))))))


;;;
;;; Function : rename
;;;
;;; Input : c - list with variables to be renamed.
;;; Output : list - renamed list
;;;
;;; Renames all the variables in the list c to new variables using make-var.
;;;


(defun rename (c)
  (let ((rlist nil))
    (labels ((rename-clause (c)
               (if c (if (listp c) (cons (rename-clause (car c)) (rename-clause (cdr c)))
                         (if (is-var c)
                            (caadr (cond ((member c rlist))
                                         ((setq rlist (append (list c (list (make-var)))
                                                              rlist)))))
                            c)))))
      (rename-clause c))))


;;;
;;; Function : resolve
;;;
;;; Input : goal - current goal to prove
;;; lengoal - length of the current goal (used for cutting)
;;; Output : t or nil or number - if the goal was proven
;;;
;;; Tries to prove the goal. If it succeeds then t is returned. If not nil is
;;; returned. If a cut occurs then a number is passed back telling how far to cut back.
;;;


(defun resolve (goal lengoal)
  (cond ((not goal) t)
        ((and (listp (car goal)) (functionp (caar goal)) (not (equal (caar goal) 'bif-value)))
         (let ((func (caar goal)))
           (cond ((equal func 'bif-cut)
                  (cond ((cutit (resolve (cdr goal) (1- lengoal)) (1- lengoal))) (lengoal)))
                 ((equal func 'bif-or)
                  (loop with ret    nil
                        for x in (cdar goal)
                        do (setq ret (resolve (cons x (cdr goal)) lengoal))
```

```
                    (if ret (return ret))))
              ((equal func 'bif-and) (resolve (append (cdar goal) (cdr goal))
                                              (+ (length (cdar goal)) (1- lengoal))))
              (t (let ((mgu (funcall func (cdar goal))))
                   (if mgu (if (listp mgu)
                               (resolve (rename (sub (cdr goal) mgu)) (1- lengoal))
                               (resolve (cdr goal) (1- lengoal)))))))))
        ((loop with retval = nil
               for clause in (cadr (member (pred-name (car goal)) hypothesis :test 'equal))
               do (setq retval
                        (let ((mgu (unify (car clause) (car goal))))
                          (if mgu (cutit (resolve (rename (sub (append (cadr clause)
                                                                       (cdr goal)) mgu))
                                                  (+ (caddr clause) (1- lengoal)))
                                          lengoal))))
               if retval return retval))))


;;;
;;; Function : set-make-var
;;;
;;; Input : none
;;; Output : none
;;;
;;; Sets the current syms value as the maximum syms value.
;;;


(defun set-make-var () (if (> syms maxsyms) (setq maxsyms syms)))


;;;
;;; Function : sub
;;;
;;; Input : c - list to be changed
;;; sublist - list of changes to be made
;;; Output : none
;;;
;;; Takes the sub list which looks like (old1 (new1) old2 (new2) ... ) and changes
;;; the variables in list with the new values.
;;;


(defun sub (c sublist)
  (if c (if (listp c)
            (cons (sub (car c) sublist) (sub (cdr c) sublist))
            (let ((x (member c sublist))) (if x (caadr x) c)))))
```

END

DATE

FILMED

DTIC

JULY 88